Displaying linear programs and their solutions with XML and SVG

Hans Chang
TechByte Limited, PO Box 39025, Christchurch, NZ
hans@tech-byte.co.nz

John F. Raffensperger
Dept. of Management, Univ. of Canterbury, Priv. Bag 4800, Christchurch, NZ
john.raffensperger@canterbury.ac.nz

Neville Churcher
Dept. of Computer Science, Univ. of Canterbury, Priv. Bag 4800, Christchurch, NZ
neville@cosc.canterbury.ac.nz

Abstract

We describe a modelling language, called XML:LP, for presenting linear and integer programs in XML, with related tools for displaying models and their solutions in web browsers. Our tools dynamically generate different model or solution views by applying different style sheets to the same model file. Graphical views help users comprehend solution and sensitivity information. We use SVG for the graphical view. We describe Java and XSLT tools to convert the popular MPS format and model solutions to and from XML:LP.

Keywords: linear programming, visualisation, XML

1 In the beginning, there was MPS

Invented 50 years ago by George Dantzig, the linear program (LP), and its difficult sibling the integer program (IP), have wide applicability in modelling real-world problems. LP and IP applies to virtually every kind of decision-making, including purchasing, vehicle routing, inventory planning, financial planning, and scheduling.

An LP is a system of linear equations with an objective function. The variables correspond to decisions, and the equations correspond to decision constraints. An IP has variables constrained to discrete values. Real-world LPs can have millions of variables and constraints. The large numbers of variables and constraints make LPs interesting as a visualisation problem.

Furthermore, the output from the solution algorithms contains an enormous amount of sensitivity analysis information, such as the marginal value (dual price) of a resource. We want to be able to visualise and examine the nature of this sensitivity information.

Typically, a commercial solver displays an LP in a text-based format such as that shown in Figure 1. Users must often truncate variable names because of software limitations, to the point where large models can be difficult to comprehend. Similarly, solvers present solutions as text, as in Figure 4 and Figure 5.

In this paper, we focus on the display of generic LP and IP models and their solutions, and the file format for exchanging these models between tools.

The standard file format of LP data is MPS, introduced in the 1960s by IBM for its early MPS/360 system. MPS is now a de facto standard among most commercial LP codes. MPS is an ASCII card image format. Its essential elements are records for the coefficients of the objective and constraints. Figure 2 shows an example of MPS. While MPS provides a de facto standard for representing LPs and IPs, no such standard exists for solutions. Each solver tool produces similar data, but in a custom format. Our approach allows such custom formats to be parsed and transformed into a well-defined XML-based format.

A consistent combined representation of both problem and solution is an essential ingredient for effective interpretation of the solution. The user needs to identify the most sensitive variables, the variables with the most freedom, etc., to make decisions based on the solution.

The simplest way XML can help is by making it easier to work with what is essentially the same MPS data. One of our contributions is the development of an XML representation for MPS. We converted the MPS file in Figure 2 to XML in Figure 10. This contribution alone would allow software to represent LPs in a consistent and modern way, and to validate those LPs.

To replace MPS, we describe a language called XML:LP. XML:LP includes the functions of MPS, with additional facility for storing solution and sensitivity information. The new format serves as a model database. Users can view the information with the tools provided in this research. We do not intend XML:LP for humans to write, edit, or read. Rather, it is to facilitate exchange between systems, as MPS does, add features missing from MPS, and to allow visualisation with standard XML software.

In XML:LP, users can create custom tags when needed, unlike MPS, which has a rigid format. LPs declared in XML can be converted to other text-based languages using XSLT. In particular, XSLT can convert the XML:LP models to Scalable Vector Graphics (SVG) format for presenting the LP graphically.

XML:LP enables LP problem and solution data to be shared and communicated between individuals and software. Our approach has been to generate web pages that can convey information about the LP problem together with information about its solution.

Since the problem and its solution may be very large, we have also had to consider how best to emphasise aspects such as the most important variables, the ranges a variable may be changed through without affecting the optimal values, and other sensitivity analysis information.

We anticipate that conversion to XML:LP would be done at application level, so the format can serve practitioners, not just those familiar with XML. The ability to represent LPs and IPs should fulfil the needs for the majority of OR practitioners. Academics may find the ability to display LP information graphically useful, as some concepts or ideas are easier to convey visually.

2 A demonstration of XML:LP

The figures below show how XML:LP can help visualise the model and the solution, in text and graphic modes.

```
LP Name: Blending
(1) MAX 18.4 RG + 3.6 HF - 7.3 BT - 12.5 HN - 18.2

(2)   BT <=   1000

(3)   RG + HF <=   12000

(4) - 94 RG + 120 BT + 74 HN + 100 CR >=   0

(5) - 11 RG + 60 BT + 4.1 HN + 2.6 CR <=   0

(6) - 17 RG + 105 BT + 12 HN + 3 CR >=   0

(7)   RG - BT - HN - CR =    0
```

Figure 1. *Problem View.XSL* applied to XML:LP.

```
NAME Blending
ROWS
  N 1
  L 2
  L 3
  G 4
  L 5
  G 6
  E 7
COLUMNS
    RG        3            1.0000000
    RG        4          -94.0000000
    RG        7            1.0000000
    RG        6          -17.0000000
    RG        5          -11.0000000
    RG        1           18.3999996
    HF        3            1.0000000
    HF        1            3.5999999
    BT        4          120.0000000
    BT        1           -7.3000002
    BT        7           -1.0000000
    BT        6          105.0000000
    BT        5           60.0000000
    BT        2            1.0000000
    HN        5            4.0999999
    HN        1          -12.5000000
    HN        4           74.0000000
    HN        6           12.0000000
    HN        7           -1.0000000
    CR        6            3.0000000
    CR        5            2.5999999
    CR        4          100.0000000
    CR        7           -1.0000000
    CR        1          -18.2000008
RHS
    RHS       2            1000.0000
    RHS       3           12000.0000
ENDATA
```

Figure 2 MPS source file, CH09B

XML:LP has two main parts, *ROWS* and *COLUMNS*. The *ROWS* element stores objective and constraint information. The *COLUMNS* section has <Continuous>, <Integer> and <Binary> sub-sections to store the different types of variables.

Our XSLT files generate the problem and solution views dynamically. Figure 1 shows a text-based view of an LP, similar to that of a standard Lindo model, as generated by the file, *Problem View.xsl*. The model is a simple LP (CH09B) that comes with Lindo, a commercial LP solver.

Figure 2 shows an MPS file for the same model.

Figure 3 shows the same LP as for Figure 2, but in XML:LP format.

Figure 4 and Figure 5 show the model's solution and sensitivity analysis in text form. These views are similar to that given by a commercial solver such as Lindo. Such text formats can be hard to interpret, without the ability to organize the information beyond what the vendor's solver provides.

```
<?xml version="1.0" encoding="utf-8" ?>
- <PROBLEM name="Blending">
  + <creation>
  + <statistics>
  - <MATRIX>
    + <SIZES>
    - <ROWS>
      - <obj name="1">
          <sense>MAX</sense>
          <best-solution>43328.84</best-solution>
        </obj>
      - <row name="2">
          <type>LT</type>
          <rhs>1000.0000</rhs>
          <dual-price>0.128844</dual-price>
          <max-increase>650.551758</max-increase>
          <max-decrease>1000.000000</max-decrease>
        </row>
      + <row name="3">
      + <row name="4">
      + <row name="5">
      + <row name="6">
      + <row name="7">
      </ROWS>
    - <COLUMNS>
      - <Continuous>
        + <column name="RG">
        - <column name="HF">
            <nz row="3">1.0000000</nz>
            <nz row="1">3.5999999</nz>
            <optimal-value>4729.704102</optimal-value>
            <reduced-cost>0.000000</reduced-cost>
            <max-increase>0.017722</max-increase>
            <max-decrease>2.084615</max-decrease>
          </column>
        + <column name="BT">
        + <column name="HN">
        + <column name="CR">
        </Continuous>
      </COLUMNS>
    </MATRIX>
  </PROBLEM>
```

Figure 3. XML source file, CH09B

Optimal Solution Value = 43328.84

Variable name	Optimal value	Reduced cost	Current coefficient	Allowable decrease	Allowable increase
RG	7270.30	0.00	18.40	.02	2.08
HF	4729.70	0.00	3.60	2.08	.02
BT	1000.00	0.00	-7.30	.13	INFINITY
HN	2446.99	0.00	-12.50	.05	9.03
CR	3823.30	0.00	-18.20	.03	2.71

Figure 4. *Solution View.XSL* applied to XML:LP, solution information

Row	Dual price	Current RHS	Allowable decrease	Allowable increase
2	.13	1000.00	1000.00	650.55
3	3.60	12000.00	4729.70	INFINITY
4	-.20	0.00	116028.99	61000.00
5	.26	0.00	15021.94	40030.00
6	0.00	0.00	INFINITY	22238.78
7	-1.56	0.00	1867.94	598.88

Figure 5. *Solution View.XSL* applied to XML:LP, constraint information

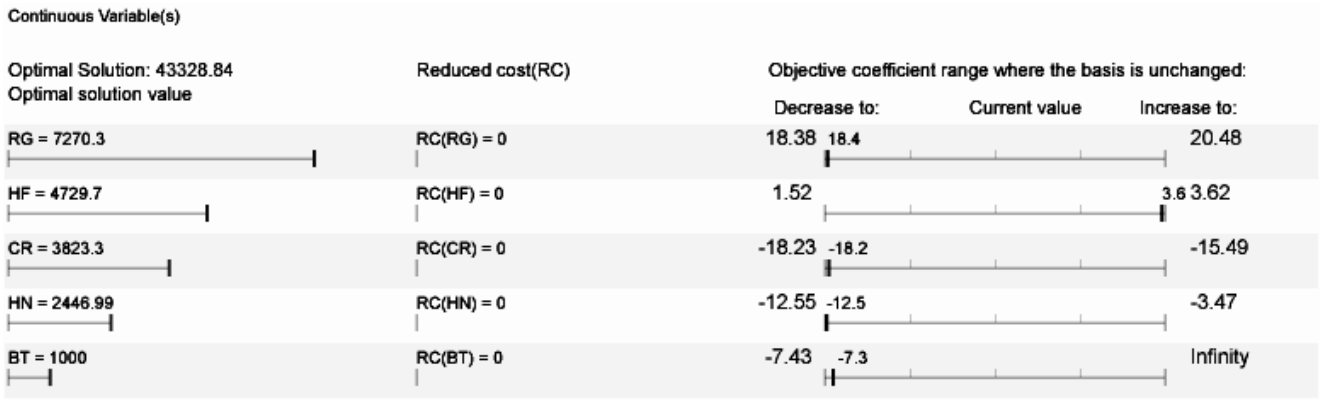

Figure 6. Source file with *SVG Solution.XSL* applied, sorted descending by optimal value. Compare to Figure 4.

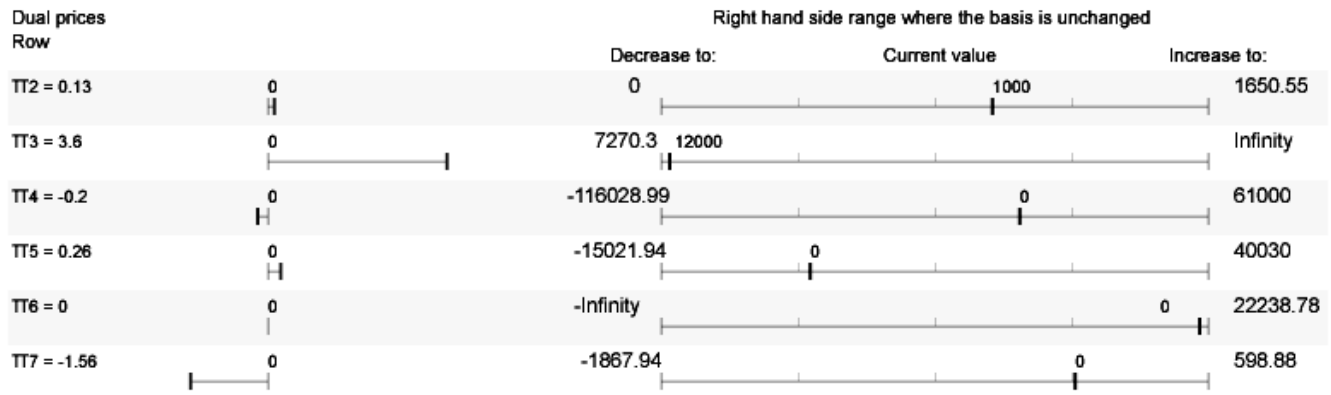

Figure 7. Source file with SVG Dual.XSL applied. Compare to Figure 5.

Figure 6 and Figure 7 indicate how the current version of our system achieved a simple visualisation. Files *SVG_Solution.XSL* and *SVG_Dual.XSL* dynamically generate graphical views of variables and constraints. We use simple visual representations to minimise visual clutter and distraction, as suggested by Tufte (1983).

SVG, an XML-based vector graphics format, allows these simple yet effective visualisations to be delivered to the desktop of users anywhere on the Internet. We take advantage of XML's ability to change, chameleon-like, into a different form, SVG in this case. The display will be easily understandable to an operations researcher.

By displaying quantities as lengths, and by allowing the user to sort information easily, we have fundamentally changed the nature of the visualisation.

We chose to show quantities as both values and lengths, for maximum information. Trade-offs had to be made, such as in Figure 6, for variable BT. The upper bound on the objective coefficient range is infinity, so the value is shown as near the far left. Unfortunately, the screen shot here does not show subtle grey bars, which help the eye follow from left to right. The user can change the format easily with only a different style sheet.

3 Literature review

This paper presents material from Chang (2003), the first author's master's thesis. At least three other parties are working on XML for math programming. Maximal-USA is working on OptML for a commercial product (Halldorsson, Thorsteinsson & Kristjansson 2000). OptML has some redundancy, as its SIZES element has information similar to that in its STATISTICS element. It has inconsistent terminology, as variables are denoted as "variables" in the STATISTICS element, but "columns" in the SIZES element. Our XML:LP is a modified form of OptML, to reduce redundancy and add features.

Lopes & Fourer (2001) are developing SNOML, an XML format primarily for stochastic programming. SNOML models have an objective function sense of minimising, thus the root element of minimize. Inside of the minimize element, there can be one or more scenario elements which contain the structure of a subproblem.

Work similar to ours is that of Fourer, Lopes, and Martin (2002). Fourer gives a nice discussion on senna.iems.nwu.edu/xml. See also Martin's (2002) remarkable and interesting work on modelling with XML. Our visualisation methods could work with their format.

Ezechukwu and Maros (2003) are developing separate mark-ups for models and solutions (where as our format combines them), with syntax for constraint programming, protocol for web services, and a Java recommendation for locating and invoking solvers over the Internet.

The chief distinction between these efforts and ours is that we focus on presentation and visualisation, to study the ability to interact with a solver through a browser.

In place of a new XML format, the reader may wonder why we did not choose Math ML. As defined by W3C, MathML is an XML application for describing mathematical notation and capturing both its structure and content. Since LPs are simultaneous equations, it is possible to write an LP in MathML format using matrix notation. However, MathML is not designed for storing or presenting LPs, and it has no facilities to indicate various LP properties, e.g. type of the variables, continuous, integer, or binary. MathML's format is less user-friendly, as the tag names are not standard LP terminologies. XML:LP preserves the LP's structure and the tag names are familiar to MS/OR practitioners.

4 XML:LP schema

XML has two ways to define a model structure, Document Type Definition (DTD) for XML, or XML Schema. DTD for XML is a cut-down version of the DTDs used for Standard Generalised Mark-up Language (SGML), which is different from XML in syntax. On the other hand, the W3C created XML Schema for constraining XML documents.

The root element of the XML:LP is *PROBLEM*. It has three complex type elements, *creation*, *statistics*, and *MATRIX*, since complex type elements have several simple types. The *MATRIX* is the major element in XML:LP Schema. Element *creation* indicates that it is an optional element, whereas element *MATRIX* is required.

The *creation* element stores basic information about this file, including model source, software used, creation date and time, model author and/or company, and key words for the model. All of the elements accept only string values, except for date and time, which receive only date, and time formats respectively.

The *statistics* element stores the number of rows, columns, integer variables, nonzeroes, and the matrix density. All the information is required to be integer, except that the element *density* is a float between 0 and 1.

MATRIX is comprised of two complex type elements, *ROWS* and *COLUMNS*. The *ROWS* and *COLUMNS* constitute the heart of XML:LP.

The ROWS element has two complex type elements: *obj* and *row*. *obj* is an optional element, which stores objective function information. One of its attributes, *active*, signals to the solver whether the current objective function should be included in the solving process. A model can have any number of objective functions, but only one can be active at any given time. The attribute *short-name* acts as an alias for the attribute *name*, during the translation to other formats that require a row name to be eight characters or less, e.g. MPS format.

A model may have one or more active *row* elements. This element stores constraint information, with the required elements of *type* and *rhs*. The elements *lhs*, *dual-price*, *max-increase* and *max-decrease* are optional as they may not be available to the author of the XML:LP file. All *row* elements are float values, except for *type*, which takes only MAX or MIN to indicate the current row's sense.

To shorten the time to find a particular class of variable, columns are divided into three elements, *Continuous*, *Integer*, and *Binary*. This allows access with XPath, e.g. //MATRIX/COLUMNS/Continuous. Xpath is faster than a loop and if statements, with all variables in one element and differentiated with attributes.

Each continuous variable can have one or more non-zero elements. As with rows, a solver would be concerned only with active columns. All elements of *column* are

float values. All variables have a default lower bound of zero to impose the non-negativity condition.

Elements *reduced-cost*, *max-increase*, and *max-decrease* do not appear in *Integer* and *Binary* elements, as integer programs do not have sensitivity information. However, *lo* and *up* elements are still present in the *Integer* element. Other than these, *Integer* and *Binary* elements have the same data structure as the *Continuous* element.

5 Testing XML:LP

To test XML:LP, we wrote code to convert existing MPS models to XML:LP format. We did this conversion in two steps. First, we use a Java program, MPStrans, to wrap XML tags around the input MPS file. We call the result simple XML format. Second, we convert the simple XML format to XML:LP with a style sheet. Figure 8 shows MPStrans. Figure 9 is a schematic of the transformation. Figure 10 shows the resulting Simple XML format.

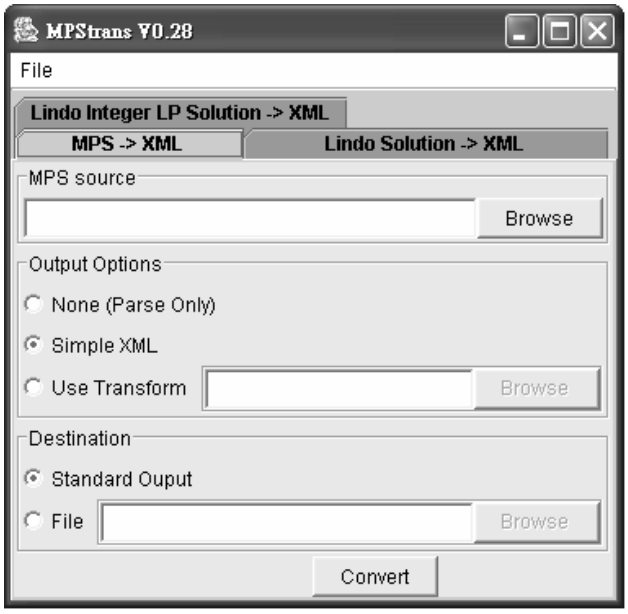

Figure 8. Screen shot of MPStrans.

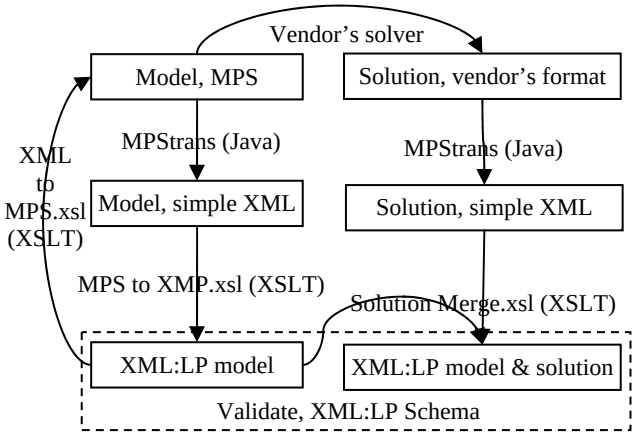

Figure 9. XML:LP transform outline

As shown in Figure 8, the MPStrans has three user inputs. (1) MPS source specifies the location of the input MPS file. MPStrans can take a number of MPS formats, fixed record fields, variable record fields, Cplex MPS extension, and Lindo MPS extensions. If the objective sense (maximising or minimising) is unspecified, then following the MPS standard, MPStrans assumes the objective is to be minimised.

(2) Output options are (a) None (Parse Only), where MPStrans only validates the structure of the MPS file; (b) Simple XML, where MPStrans wraps the MPS inputs with XML tags, and (c) Use Transform. When supplied with the "MPS to XML.xsl" style sheet, MPStrans will apply the style sheet to the simple XML document object model (DOM) and transform it to XML:LP format.

(3) Destination. MPStrans can write results to stdio.

As in Figure 10, the simple XML format tags consist primarily of MPS names. Integer variables have an attribute called *type* with a value of *integer* under *column* element. Binary variables will have an attribute value of *BV*. If a *column* element does not have a specified *type* attribute then it assumed to be a continuous variable.

```
<?xml version="1.0" encoding="utf-8"?>
<mpslp name="blending" sense="max">
<rows><row name="1" sense="n"/>
        <row name="2" sense="l"/>
        <row name="3" sense="l"/>
        <row name="4" sense="g"/>
        <row name="5" sense="l"/>
        <row name="6" sense="g"/>
        <row name="7" sense="e"/>
</rows>
<columns>
<column name="rg" row="3">1.0000000</column>
<column name="rg" row="4">-94.0000000</column>
<column name="rg" row="7">1.0000000</column>
<column name="rg" row="6">-17.0000000</column>
<column name="rg" row="5">-11.0000000</column>
<column name="rg" row="1">18.3999996</column>
<column name="hf" row="3">1.0000000</column>
<column name="hf" row="1">3.5999999</column>
<column name="bt" row="4">120.0000000</column>
<column name="bt" row="1">-7.3000002</column>
<column name="bt" row="7">-1.0000000</column>
<column name="bt" row="6">105.0000000</column>
<column name="bt" row="5">60.0000000</column>
<column name="bt" row="2">1.0000000</column>
<column name="hn" row="5">4.0999999</column>
<column name="hn" row="1">-12.5000000</column>
<column name="hn" row="4">74.0000000</column>
<column name="hn" row="6">12.0000000</column>
<column name="hn" row="7">-1.0000000</column>
<column name="cr" row="6">3.0000000</column>
<column name="cr" row="5">2.5999999</column>
<column name="cr" row="4">100.0000000</column>
<column name="cr" row="7">-1.0000000</column>
<column name="cr" row="1">-18.2000008</column>
</columns>
<rhs><rhs vector="rhs" row="2">1000.0000</rhs>
        <rhs vector="rhs" row="3">12000.0000</rhs>
</rhs></mpslp>
```

Figure 10. Resultant output using simple XML format on *CH09B.mps*.

Since the model is in XML now, we can transform it to XML:LP format by applying the style sheet *MPS to XML.xsl*. Transforming the simple XML format to any other ASCII format is also possible by applying a different style sheet. Of course, we immediately convert this simple XML format to XML:LP.

6 Construction of a parser for MPS

6.1 A grammar for MPS

MPS is a fixed-column character based format, horrible to read, but easy to parse. The most simple-minded approach to building a parser for MPS would be to write a handcrafted special-purpose program. Such a program is responsible for reading the MPS format, processing it, and translating it into the desired output formats. This approach has several potential difficulties.

Firstly, even if a formal specification exists, it is hard to ensure that a robust error-free parser has been written. Even if this is achieved, it is likely to be challenging to modify the parser when (inevitably) changes occur in the input format, processing requirements or output.

An effective alternative involves constructing a parser in several stages: (1) Obtain a grammar for the input language. (2) Specify actions to take when input strings matching grammar constructs are recognised. (3) Use a parser generator to produce a parser automatically. (4) Update grammar and/or actions as requirements evolve.

6.2 Generation

Given a set of rules and actions as its input, a parser generator will produce a parser for the corresponding language. This is then compiled and used with other supporting code. The archetypal parser generator is yacc (Johnson 1975) which generates parser written in the C language. Similarly, lexical analyser generators, such as lex, may be used in conjunction with the parser generator.

We use the CUP (constructor or useful parsers) parser generator, which produces parsers in Java and JFlex. Further details may be found in parsing texts (Appel 1998), www.cs.princeton.edu/~appel/modern/java/CUP, or www.jflex.de, and Chang (2003).

The MPSTrans software includes several distinct parsers. One parser recognises the MPS format and converts it to an appropriate XML form. Another parses Lindo's LP output, and a further parser recognises Lindo's output for integer programs. Further parsers may be added to interface to other solvers.

7 Adding solution information to XML:LP

To add the solution to the XML:LP file, we need to call a solver. However, all known solvers use a proprietary or MPS format. Therefore, we use a style sheet to convert XML:LP back to MPS format.

After transforming the XML:LP to MPS, we can feed the MPS file to any solvers that read this format. In this section, we used Lindo (2003) to generate the solution information, as shown in Figure 11.

```
LP OPTIMUM FOUND AT STEP      4

        OBJECTIVE FUNCTION VALUE
        1)      43328.84

 VARIABLE        VALUE          REDUCED COST
       RG      7270.295898          0.000000
       HF      4729.704102          0.000000
       BT      1000.000000          0.000000
       HN      2446.991455          0.000000
       CR      3823.304688          0.000000


      ROW   SLACK OR SURPLUS     DUAL PRICES
        2)         0.000000          0.128844
        3)         0.000000          3.600000
        4)         0.000000         -0.204298
        5)         0.000000          0.258835
        6)     22238.777344          0.000000
        7)         0.000000         -1.556829

NO. ITERATIONS=       4

RANGES IN WHICH THE BASIS IS UNCHANGED:
                         OBJ COEFFICIENT RANGES
VARIABLE         CURRENT        ALLOWABLE         ALLOWABLE
                   COEF         INCREASE          DECREASE
      RG        18.400000         2.084615          0.017722
      HF         3.600000         0.017722          2.084615
      BT        -7.300000         INFINITY          0.128844
      HN       -12.500000         9.033334          0.052654
      CR       -18.200001         2.710000          0.033700

                         RIGHTHAND SIDE RANGES
     ROW         CURRENT        ALLOWABLE         ALLOWABLE
                   RHS          INCREASE          DECREASE
       2      1000.000000       650.551758      1000.000000
       3     12000.000000         INFINITY      4729.704102
       4         0.000000     61000.003906    116028.992188
       5         0.000000     40030.000000     15021.935547
       6         0.000000     22238.777344         INFINITY
       7         0.000000       598.877991      1867.942139
```

Figure 11. Lindo solution to our example model.

Following solution with a solver, XSLT incorporates the solution into the XML:LP file. As with the MPS format, we need to wrap the solution with XML tags before XSLT can process the information. This is where MPStrans comes in again (Figure 12).

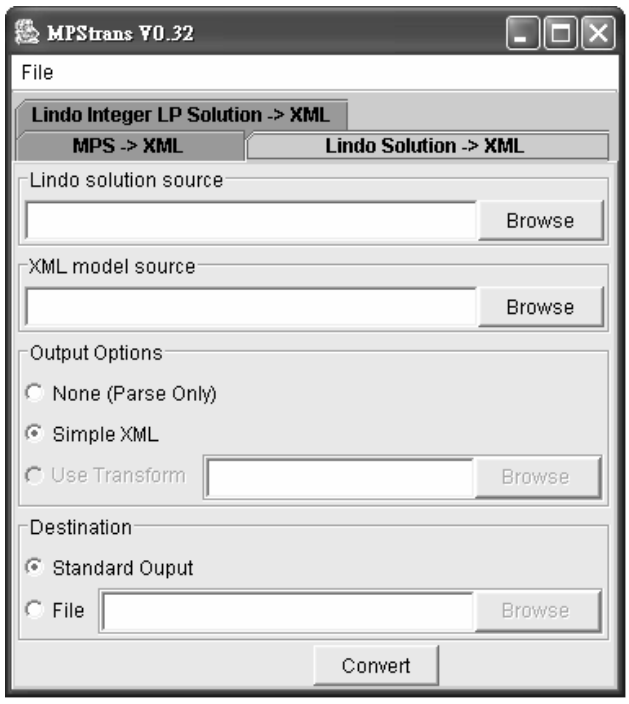

Figure 12. MPStrans, Lindo solution to XML

MPStrans has two panels for reading Lindo solution files, “Lindo Solution -> XML” and “Lindo Integer LP Solution -> XML”. These two panels have the same interface, one for the LPs and the other for integer and binary programs.

For the solution read-in, the MPStrans has four user inputs.

(1) Lindo solution source: This input specifies the location of the input Lindo solution file.

(2) XML model source: This input specifies the location of the input XML:LP model file.

```
<?xml version="1.0" encoding="utf-8"?>
<solution><constraints>
<objective><value>-43328.84</value></objective>

<constraint name="2"><dual_price>0.128844</dual_price>
<allowable_increase>650.551758</allowable_increase>
<allowable_decrease>1000.000000</allowable_decrease>
</constraint>
…
<constraint name="7"><dual_price>-1.556829</dual_price>
<allowable_increase>598.877991</allowable_increase>
<allowable_decrease>1867.942139</allowable_decrease>
</constraint>
</constraints>

<variables>
<variable name="rg"><value>7270.295898</value>
<reduced_cost>0.000000</reduced_cost>
<allowable_increase>0.017722</allowable_increase>
<allowable_decrease>2.084615</allowable_decrease>
</variable>

<variable name="hf"><value>4729.704102</value>
<reduced_cost>0.000000</reduced_cost>
<allowable_increase>2.084615</allowable_increase>
<allowable_decrease>0.017722</allowable_decrease>
</variable>

<variable name="bt"><value>1000.000000</value>
<reduced_cost>0.000000</reduced_cost>
<allowable_increase>0.128844</allowable_increase>
<allowable_decrease>infinity</allowable_decrease>
</variable>

<variable name="hn"><value>2446.991455</value>
<reduced_cost>0.000000</reduced_cost>
<allowable_increase>0.052654</allowable_increase>
<allowable_decrease>9.033334</allowable_decrease>
</variable>

<variable name="cr"><value>3823.304688</value>
<reduced_cost>0.000000</reduced_cost>
<allowable_increase>0.033700</allowable_increase>
<allowable_decrease>2.710000</allowable_decrease>
</variable>
</variables></solution>
```

Figure 13 Simple XML solution format. Constraints 3 through 6 were omitted for brevity. Compare to Figure 2

(3) Output options: (a) None (Parse Only): When the user selects this option, MPStrans validates the structure of the Lindo solution file. (b) Simple XML: When the user selects this option, MPStrans will wrap the Lindo solution inputs with XML tags. (c) Use Transform: When supplied with “Solution Merger.xsl”, MPStrans applies the style sheet to the simple XML solution document object model (DOM) and merges the solution into the model file specified in input field 2.

(4) Destination: (a) Standard Output: MPStrans writes the results to the standard on-screen buffer, e.g. command prompt. (b) File: This input specifies the desired location and filename for the output.

The XML tags in the simple XML solution format primarily consist of names used in Lindo solution format. For integer or binary solution outputs, the tags will have a type attribute with a value of “integer”. See Figure 13

Once the solution information is in XML format, we can use a XSLT processor and “Solution Merge.xsl” to incorporate the solution information.

In this style sheet, we are working with two XML documents: the model file and the solution information in simple XML format. The variables, *doc_node* and *solution*, specify the location of the solution file to the XSLT processor. The “Solution Merge.xsl” file does not process the creation and statistics elements, as they do not store solution information.

Figure 14 shows the XML:LP file with the solution information, for our example model. We tested our code for all 60 sample files that come with Lindo, which serves as a varied set of applications, as there are 15 types of models. Some of these are maximisations and some are minimisations. The optimal objective value for a maximisation changes sign, since the objective sense is changed to minimisation to conform to the MPS standard. The optimal solution values remain the same, regardless of the sign of the optimal objective function solution. Moreover, our results indicate that XML:LP format is suitable for a wide variety of applications.

The MPStrans program had trouble parsing five of Lindo’s example files. These five use sense indicators (N, G, L, or E) or bounds indicators (LO, UP, FX, FR, MI, PL, UI or BV) as variable names. While this does not violate the rules set by MPS, it upset the grammar used in MPStrans and there is no easy fix to it. For these files, if we change the names of the offending variables, the code works fine.

8 Generating graphical views of LPs

Thus far, we have shown the data structure of XML:LP and how to transform MPS files into XML:LP using XSLT. Now, we need a way to visualise the data stored in XML:LP. There are three ways of rendering XML documents, using Cascading Style Sheets (CSS), XML Style sheet, or XML Style sheet with CSS.

CSS was originally created for Hyper Text Mark-up Language (HTML), to separate formatting from content. It has the ability to control the font property, including font family, size, colour and styling and the page layout, including paragraph margins and padding around tables. Web developers use CSS to specify on-screen positioning of elements. However, CSS lacks XSLT’s ability to filter

elements and process the document according to given rules. On the other hand, XSLT does not have the formatting ability of CSS. Thus, we will be using XSLT to process the document, then applying the formatting with CSS, the best from both worlds.

```
<?xml version="1.0" encoding="utf-8" ?>
- <PROBLEM name="Blending">
  + <creation>
  + <statistics>
  - <MATRIX>
    - <ROWS>
      - <obj name="1" active="1">
          <sense>MAX</sense>
          <best-solution>-43328.84</best-solution>
        </obj>
      - <row name="2" active="1">
          <type>LT</type>
          <rhs>1000.0000</rhs>
          <dual-price>0.128844</dual-price>
          <max-increase>650.551758</max-increase>
          <max-decrease>1000.000000</max-decrease>
        </row>
      + <row name="3" active="1">
      + <row name="4" active="1">
      + <row name="5" active="1">
      + <row name="6" active="1">
      + <row name="7" active="1">
      </ROWS>
    - <COLUMNS>
      - <Continuous>
        - <column name="RG" active="1">
            <nz row="3">1.0000000</nz>
            <nz row="4">-94.0000000</nz>
            <nz row="7">1.0000000</nz>
            <nz row="6">-17.0000000</nz>
            <nz row="5">-11.0000000</nz>
            <nz row="1">18.3999996</nz>
            <optimal-value>7270.295898</optimal-value>
            <reduced-cost>0.000000</reduced-cost>
            <max-increase>0.017722</max-increase>
            <max-decrease>2.084615</max-decrease>
          </column>
        + <column name="HF" active="1">
        + <column name="BT" active="1">
        + <column name="HN" active="1">
        + <column name="CR" active="1">
        </Continuous>
      </COLUMNS>
    </MATRIX>
  </PROBLEM>
```

Figure 14. XML:LP with the solution information.

In this research, we provide four ways of presenting the XML:LP model: text based model view, text based solution view, graphical variables view, and graphical constraints view. XSL dynamically generates each view when the XML:LP model and the respective style sheets pass through the XSL processor. The browser then formats the output with the associated CSS file.

It is possible to use Saxon as the XSL processor, however it would be too laborious to enter the commands explicitly when each view is required. Moreover, the user needs to call an external viewer to view the content, e.g. Microsoft Internet Explorer (IE). It is more sensible to use the browser's internal XSL processor to carry out the transformation and to render the output, a technique known as client side transformation. Currently, only the latest browsers, e.g. Internet Explorer 6, Netscape 6+, and Mozilla 1.0, have the proper support for transforming XML documents. We chose IE 6 in this research as it has the biggest share of user base and the best support for various ways of transformation.

Unfortunately, the client side transformation protocol is proprietary and limited to IE only. This is because W3C has not specified a standard way to carry out transformation in a browser. The future release of XSLT specification will remedy this problem, but for now, client side transformation is limited to Microsoft Internet Explorer 6 or more specifically, IE6.0's build in XSL processor. However, there are no proprietary functions in the proposed XSLT style sheet. This is evident from the fact that we can use Saxon, a XSL processor that follows W3C recommendation, and the output can be viewed on any W3C compatible viewers.

In addition to the traditional text based solution view, we propose to use graphics as an alternative medium. We believe that graphical view gives the viewer better recognition of the magnitude of the solution values. Sensitivity information, especially the allowable increase and decrease, may be easier for to read using graphs instead of numbers.

At this point, we need to choose the image's format, raster or vector based. It is possible to use raster-based images, e.g. GIF or JPEG. However, these formats can hog bandwidth. The printing resolution of the raster image is often limited to 70 dots per inch, while the surrounding text would typically be 300, 600 or more dots per inch. Most people find the raster images' print quality to be unacceptable. Due to these deficiencies, we ruled out the use of raster-based images and looked at vector-based images.

SVG is ideal for vector graphics. It is an XML based language defined by W3C, currently at version 1.1. SVG graphics are a significant step forward in terms of image quality and overall file size, as it uses vector information instead of pixels and is written in plain text mark-up. This makes editing and reusing a lot easier, as the user can directly edit ASCII text. The browser can magnify images with no loss of quality. Only the printer capability limits print quality. Finally, we can generate SVG graphics using XLST; there is no need for any complicated software tools.

Currently, only a handful of browsers have native support for SVG graphics. Users may need to download a plug-in for their browser. One such plug-in can be freely downloaded from http://www.adobe.com/svg/.

Two files generate the text based model view, *Problem View.xsl* and *LP Format.css*. The XSLT extracts the model information from the XML:LP file and the CSS formats the page.

The output from the style sheet is an XML document with three major elements, *ROWS, RANGES, and TYPES*. The browser then takes the XML document and CSS file to render the page on screen. All of the coefficients shown

on the screen were rounded to two decimal places to avoid cluttering the display.

In Figure 4 and Figure 5, the user can sort views easily, by variable name, optimal value, etc. For example, when the user clicks “optimal value” once, the browser will sort the optimal values ascending. On the second click, the table will do a descending sort. With the sorts, there are 10 different main options for viewing the problem and solution, available to the user via a drop-down menu.

In Figure 6 and Figure 7, the first thing the style sheet does is to determine the width and height of the image. The width is currently set to 820 pixels. The number of active variables determines the height; each active variable adds 32 pixels to the height. After the size of the SVG graph is determined, the style sheet writes the graph labels, e.g., “Reduced Cost RC()”. The template *draw_bar* in *svg_bar.xsl* draws the variable and reduced cost graphs. The *draw_range* template in *svg_range.xsl* draws the range graph. The *graph.css* controls the styling of the graphs, including colouring, line thickness, and font properties. For the solution value, we use the xsl:sort function to determine the maximum solution value among all active variables. Then the unit/pixel relationship is calculated by dividing the length of the solution view, *$x1_e - $x1_s*, by the maximum solution value. For the reduced cost value, we repeat the same steps; however, this is limited to continuous variables only. Next, the style sheet calls the corresponding templates to draw the bar and range graphs for each variable.

Each variable’s range graph is normalised with respect to the range, i.e. “increase to” to “decrease to”. If either the lower or upper bound has a value of infinity then the indicator will be placed at 5 pixels from the other end. There is no range graph for *CH18Bs.xml*, as it is a mixed integer LP. After the SVG file has been generated, it is rendered with *graph.css* in an SVG viewer, e.g. Internet Explorer.

Our visual display uses colour to good effect. For example, a continuous variable has a black text and line, an integer variable has a blue text and line, and a binary variable has a red text and line.

The graphical constraint view has a dual price view and a right hand side range view. The right hand side range view is limited to LPs, as integer or binary programs do not have valid sensitivity information. The graph.css style sheet renders the graph above.

The *SVG_Dual.xsl* has the same initialisation as the *SVG_Solution.xsl*, including setting the width and height of the SVG graphics and writing out the text labels. Each additional row adds another 32 pixels to the height of the graph. The same CSS file, graph.css, sets the format.

The xsl:sort function calculates the maximum and minimum dual prices for the *row* elements. When calculating the pixel/(dual unit), we need to be careful with the signs of the maximum and minimum dual prices, thus a *choose* statement is used. When the minimum dual price equals zero and the maximum dual price is greater than zero, the dual price range is the value of the maximum dual price. Otherwise, the dual price range is calculated by subtracting the minimum dual price from the maximum dual price. Range information is not produced for integer programs, which lack this output. In addition to the linearity condition, the range graph is called only if the maximum and minimum dual prices are non-zero.

The bar graph has two parts, text labels and graphs. The *svg_bar.xsl* does not have any hard coded x-y coordinates, instead it relies on its caller for this information, e.g., *SVG_Solution.xsl* or *SVG_Dual.xsl*.

The text label writes the variable name, with π in front when processing dual price and with RC(name) format when processing reduced cost.

The bar graph consists of three parts: starting marker (|), middle line (-), and ending marker (|). The middle line and ending mark are drawn only if the solution value, reduced cost, or dual price is non-zero. The ending mark is a thicker marker, and is omitted if the length of the middle line is less than three pixels on screen.

When the solution value and reduced cost or dual price are all greater than 0 then the origin of the graph starts at left hand end. The graph then extends towards the right. When all the values are less or equal to zero, the origin of the graph shifts to the right and extends toward the left. Zero is added to the origin of the graph to help readers identify the origin. When the values are mixed, there is no fixed origin point. Instead, it is calculated with respect to the maximum and minimum values. Similar work is done for the Range graph generation.

9 Conclusion

We have defined an XML format for mathematical models, called XML:LP. XML:LP can accommodate a variety of MS/OR models, linear, integer, binary and mixed integer LPs. XML:LP can store the solution, and XML:LP files can be validated against the included XML Schema.

To smooth the transition to XML:LP, this research has developed tools to convert XML:LP to and from MPS. The Java program, *MPStrans*, is used to wrap XML tags around the input MPS. An XSLT then transforms the MPS file from simple XML format to XML:LP. We can modify the output by changing the structure of the XSLT file, instead of editing and recompiling the Java program. By using the included style sheet, *XML to MPS.xsl*, users can convert XML:LP back to MPS. We used Lindo 6.1 to generate the solution and the same Java program to wrap XML tags around it. Another style sheet, *Solution Merge.xsl*, embeds the XML:LP file. This concludes one cycle of transformation of MPS to XML:LP to MPS transformation.

XML:LP appears to be efficient at storing real-life sized models, especially when compressed. We tested only a small number of large models (11,000 to 48,000 non-zeroes), but found that XML:LP file sizes were almost exactly the same size as MPS files sizes before compression, and smaller than MPS when compressed.

The beauty of XML:LP is the ability to generate model and solution views dynamically. This saves the redundancy of having a file for each view. We provided four style sheets to generate four different views: text based model and solution views, graphical variable and constraint views. The graphical views can help users to understand the sensitivity information better. In the graphical views, we used SVG for its superior print quality and smaller file size.

In conclusion, this research has provided the tools to generate and render XML:LP models visually. However, we have not touched on solver integration and server side process. Until these two steps are realised, XML:LP will not be popular among MS/OR users. Future work may want to look at how to accomplish these tasks, so the users can benefit from XML technology.

10 References

Appel, A. W., *Modern Compiler Implementation in Java*, Cambridge University Press, 1998.

Chang, Hans, *Modelling and Presenting Mathematical Programs with XML:LP*, Master's Thesis, University of Canterbury, 2003.

Ezechukwu, O.C., & Maros, I., "OOF: Open Optimization Framework," Technical Report ISSN 1469-4174, Imperial College London, UK, Apr 2003, http://www.doc.ic.ac.uk/old-doc/deptechrep/DTR03-7.pdf.

Fourer, R., Lopes, L. & Martin, R.K. "A W3C XML schema for linear programming," Technical report, Northwestern Univ. and Univ. of Chicago, 2002.

Halldorsson, B.V., Thorsteinsson, E.S., & Kristjansson, B., "A modeling interface to non-linear programming solvers an instance: xMPS, the extended MPS format," Technical report, Carnegie Mellon University and Maximal Software, 2000.

Johnson, S.C., "Yacc, yet another complier compiler," Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, NK, 1975.

Lindo Systems, Inc., 2003, http://www.lindo.com.

Lopes, L. & Fourer, R., "An XML-based Format for Communicating Optimization Problems," (SNOML) http://senna.iems.northwestern.edu/xml/presentations/LopesFourerMiamiBeach01, Feb 2001.

Martin, R. Kipp, "A Modeling System for Mixed Integer Linear Programming Using XML Technologies," working paper, Graduate School of Business, Univ. of Chicago, 11 Dec 2002, revised 27 Feb 2003.

SMPS format online documentation, 13 Nov 2002, www.mgmt.dal.ca/sba/profs/hgassmann/SMPS2.htm.

Tufte, Edward, *The Visual Display of Quantitative Information*, Graphics Press, Cheshire, Conn., 1983.

Intelligent Mathematical Programming System, 29 Nov 1998, carbon.cudenver.edu/~hgreenbe/imps/imps.html.

XPath W3C Recommendation Version 1.0., 16 Nov 1999 http://www.w3c.org/TR/xpath.